15 Decoupling your infrastructure: Elastic Load Balancing and Simple Queue Service

This chapter covers

- The reasons for decoupling a system
- Synchronous decoupling with load balancers to distribute requests
- Hiding your backend from users and message producers
- Asynchronous decoupling with message queues to buffer message peaks

Imagine that you want some advice from us about using AWS, and therefore we plan to meet in a cafe. To make this meeting successful, we must

- Be available at the same time
- Be at the same place
- Find each other at the cafe

The problem with our meeting is that it's tightly coupled to a location. We live in Germany; you probably don't. We can solve that issue by decoupling our meeting from the location. So we change plans and schedule a Google Hangout session. Now we must:

- Be available at the same time
- Find each other in Google Hangouts

Google Hangouts (and other video/voice chat services) does synchronous decoupling. It removes the need to be at the same place, while still requiring us to meet at the same time.

We can even decouple from time by using email. Now we must:

- Find each other via email

Email does asynchronous decoupling. You can send an email when the recipient is asleep, and they can respond later when they're awake.

Examples are 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

NOTE To fully understand this chapter, you'll need to have read and understood the concept of auto-scaling covered in chapter 14.

So far you learned about two ways to decouple a meeting:

- *No decoupling*—We have to be at the same place (the cafe), at the same time (3 p.m.), and find each other (I have black hair and I'm wearing a white shirt).
- *Synchronous decoupling*—We can now be at different places. But still, we have to find a common time (3 p.m.), and find each other (exchange Skype IDs).
- *Asynchronous decoupling*—We can be at different places and now also don't have to find a common time. We only have to find each other (exchange email addresses).

A meeting isn't the only thing that can be decoupled. In software systems, you can find a lot of tightly coupled components:

- A public IP address is like the location of our meeting: To make a request to a web server, you must know its public IP address, and the virtual machine must be connected to that address. If you want to change the public IP address, both parties are involved in making the appropriate changes.

- If you want to make a request to a web server, the web server must be online at the same time. Otherwise your request will be rejected. There are many reasons why a web server can be offline: someone might be installing updates, a hardware failure, and so on.

AWS offers a solution for synchronous and asynchronous decoupling. The *Elastic Load Balancing* (ELB) service provides different types of load balancers that sit between your EC2 instances and the client to decouple your requests synchronously. For asynchronous decoupling, AWS offers a *Simple Queue Service* (SQS) that provides a message queue infrastructure. You'll learn about both services in this chapter. Let's start with ELB.

15.1 Synchronous decoupling with load balancers

Exposing a single EC2 instance running a web server to the outside world introduces a dependency: your users now depend on the public IP address of the EC2 instance. As soon as you distribute the public IP address to your users, you can't change it anymore. You're faced with the following issues:

- Changing the public IP address is no longer possible because many clients rely on it.
- If you add an additional EC2 instance (and IP address) to handle the increasing load, it's ignored by all current clients: they're still sending all requests to the public IP address of the first server.

You can solve these issues with a DNS name that points to your server. But DNS isn't fully under your control. DNS resolvers cache responses. DNS servers cache entries, and sometimes they don't respect your time to live (TTL) settings. For example, you might ask DNS servers to only cache the name-to-IP address mapping for one minute, but some DNS servers might use a minimum cache of one day. A better solution is to use a load balancer.

A load balancer can help decouple a system where the requester awaits an immediate response. Instead of exposing your EC2 instances (running web servers) to the outside world, you only expose the load balancer to the outside world. The load balancer then forwards requests to the EC2 instances behind it. Figure 15.1 shows how this works.

The requester (such as a web browser) send an HTTP request to the load balancer. The load balancer then selects one of the EC2 instances and copies the original HTTP request to send to the EC2 instance that it selected. The EC2 instance then processes the request and sends a response. The load balancer receives the response, and sends the same response to the original requester.

AWS offers different types of load balancers through the Elastic Load Balancing (ELB) service. All load balancer types are fault-tolerant and scalable. They differ mainly in the protocols they support:

- *Application Load Balancer (ALB)—HTTP, HTTPS*
- *Network Load Balancer (NLB)*—TCP
- *Classic Load Balancer (CLB)*—HTTP, HTTPS, TCP, TCP+TLS

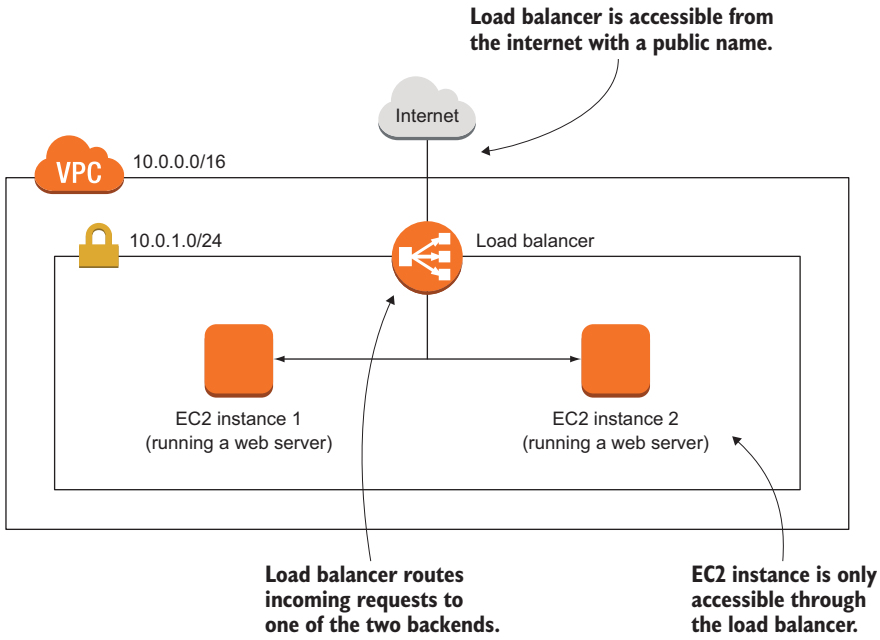


Figure 15.1 A load balancer synchronously decouples your EC2 instances.

The Classic Load Balancer is the oldest of the load balancers. If you start a new project, we recommend going with the ALB or NLB, because they are in most cases more cost efficient and more feature-rich.

> **NOTE** The ELB service doesn't have an independent Management Console. It's integrated into the EC2 Management Console.

Load balancers can be used with more than web servers—you can use load balancers in front of any systems that deal with request/response-style communication as long as the protocol is based on TCP.

15.1.1 Setting up a load balancer with virtual machines

AWS shines when it comes to integrating services. In chapter 14, you learned about auto-scaling groups. You'll now put an ALB in front of an auto-scaling group to decouple traffic to web servers, to remove a dependency between your users and the EC2 instance's public IP address. The auto-scaling group will make sure you always have two web servers running. As you learned in chapter 14, that's the way to protect against downtime caused by hardware failure. Servers that are started in the auto-scaling group will automatically register with the ALB. Figure 15.2 shows what the setup will look like. The interesting part is that the EC2 instances are no longer accessible directly from the public internet, so your users don't know about them. They don't know if there are 2 or 20 EC2 instances running behind the load balancer. Only the

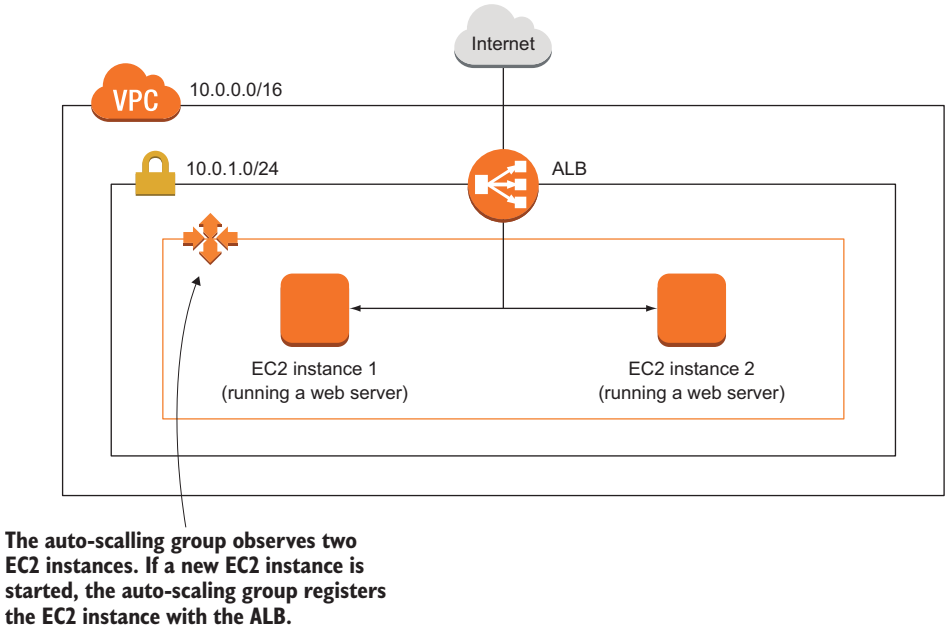


Figure 15.2 Auto-scaling groups work closely with ALB: they register a new web server with the load balancer.

load balancer is accessible and forwards requests to the backend servers behind it. The network traffic to load balancers and backend EC2 instances is controlled by security groups, which you learned about in chapter 6. If the auto-scaling group adds or removes EC2 instances, it will also register new EC2 instances with the load balancer and deregister EC2 instances that have been removed.

An ALB consists of three required parts and one optional part:

- *Load balancer*—Defines some core configurations, like the subnets the load balancer runs in, whether the load balancer gets public IP addresses, whether it uses IPv4 or both IPv4 and IPv6, and additional attributes.
- *Listener*—The listener defines the port and protocol that you can use to make requests to the load balancer. If you like, the listener can also terminate TLS for you. A listener links to a target group that is used as the default if no other listener rules match the request.
- *Target group*—A target group defines your group of backends. The target group is responsible for checking the backends by sending periodic health checks. Usually backends are EC2 instances, but could also be a Docker container running on EC2 Container Service or a machine in your data center paired with your VPC.

- *Listener rule*—Optional. You can define a listener rule. The rule can choose a different target group based on the HTTP path or host. Otherwise requests are forwarded to the default target group defined in the listener.

Figure 15.3 shows the ALB parts.

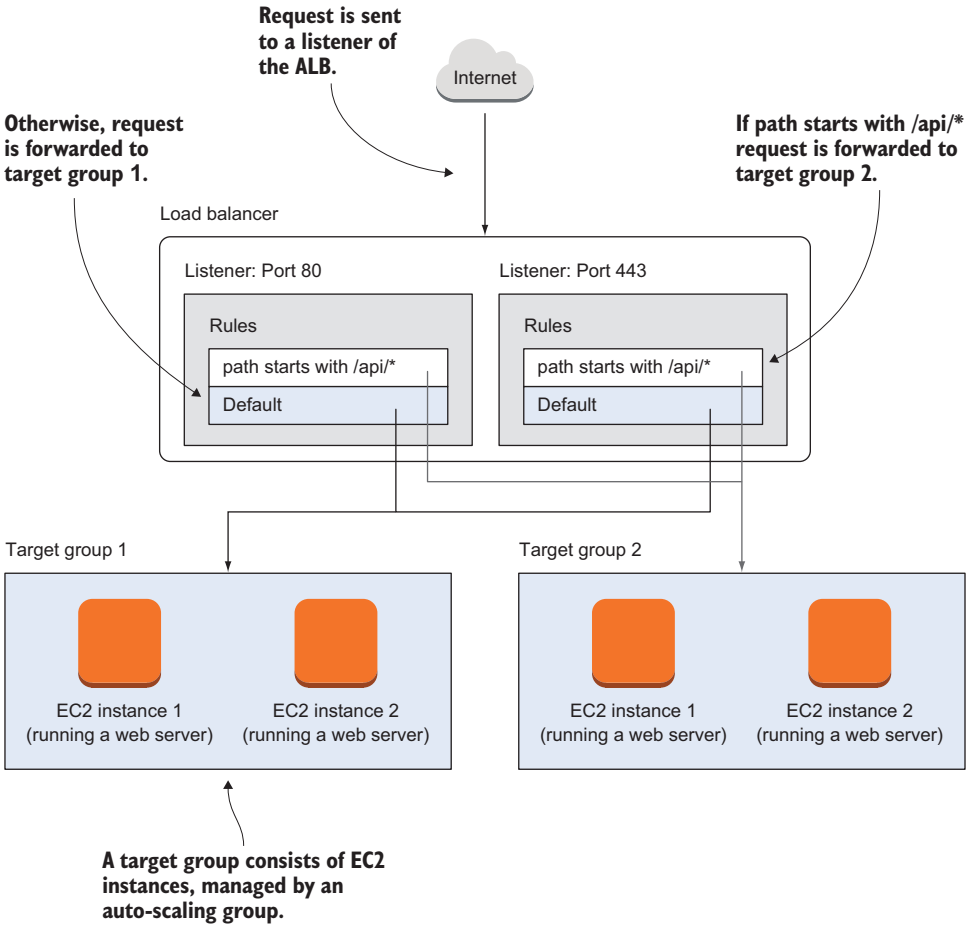
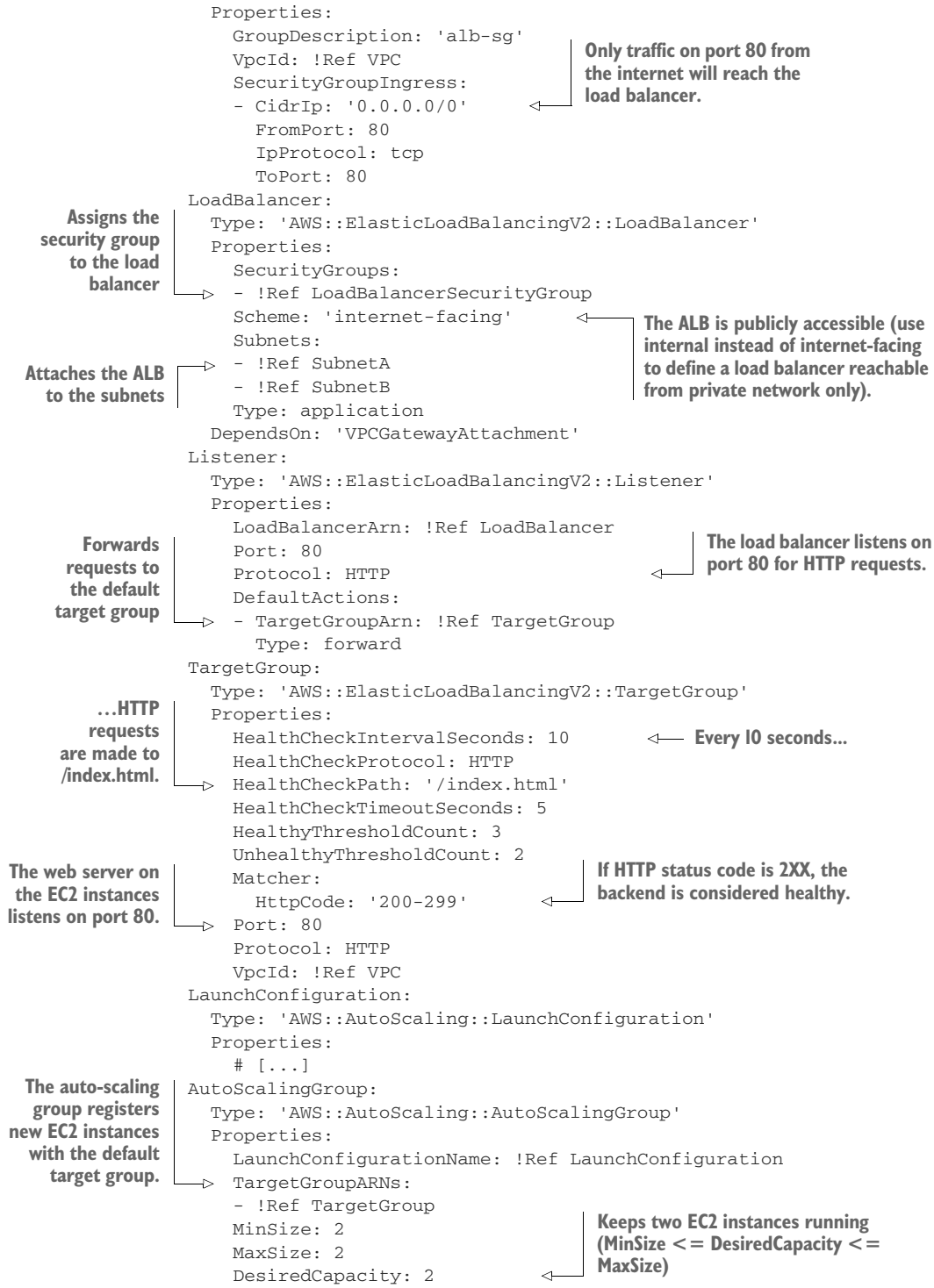


Figure 15.3 ALB consists of parts: load balancer, listener, target group, and optional listener rules.

The next listing shows a CloudFormation template snippet to create an ALB and connect it with an auto-scaling group. The listing implements the example shown in figure 15.2.

Listing 15.1 Creating a load balancer and connecting it to an auto-scaling group

```
# [...]
LoadBalancerSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
```


Properties:
GroupDescription: 'alb-sg'
VpcId: !Ref VPC
SecurityGroupIngress:
- CidrIp: '0.0.0.0/0'
FromPort: 80
IpProtocol: tcp
ToPort: 80
Only traffic on port 80 from the internet will reach the load balancer.
LoadBalancer:
Type: 'AWS::ElasticLoadBalancingV2::LoadBalancer'
Properties:
SecurityGroups:
- !Ref LoadBalancerSecurityGroup
Assigns the security group to the load balancer
Scheme: 'internet-facing'
The ALB is publicly accessible (use internal instead of internet-facing to define a load balancer reachable from private network only).
Subnets:
- !Ref SubnetA
- !Ref SubnetB
Attaches the ALB to the subnets
Type: application
DependsOn: 'VPCGatewayAttachment'
Listener:
Type: 'AWS::ElasticLoadBalancingV2::Listener'
Properties:
LoadBalancerArn: !Ref LoadBalancer
Port: 80
Protocol: HTTP
The load balancer listens on port 80 for HTTP requests.
DefaultActions:
- TargetGroupArn: !Ref TargetGroup
Type: forward
Forwards requests to the default target group
TargetGroup:
Type: 'AWS::ElasticLoadBalancingV2::TargetGroup'
Properties:
HealthCheckIntervalSeconds: 10
Every 10 seconds...
HealthCheckProtocol: HTTP
HealthCheckPath: '/index.html'
...HTTP requests are made to /index.html.
HealthCheckTimeoutSeconds: 5
HealthyThresholdCount: 3
UnhealthyThresholdCount: 2
Matcher:
HttpCode: '200-299'
If HTTP status code is 2XX, the backend is considered healthy.
Port: 80
The web server on the EC2 instances listens on port 80.
Protocol: HTTP
VpcId: !Ref VPC
LaunchConfiguration:
Type: 'AWS::AutoScaling::LaunchConfiguration'
Properties:
[...]
AutoScalingGroup:
Type: 'AWS::AutoScaling::AutoScalingGroup'
Properties:
LaunchConfigurationName: !Ref LaunchConfiguration
TargetGroupARNs:
- !Ref TargetGroup
The auto-scaling group registers new EC2 instances with the default target group.
MinSize: 2
MaxSize: 2
DesiredCapacity: 2
Keeps two EC2 instances running (MinSize <= DesiredCapacity <= MaxSize)

```
      VPCZoneIdentifier:
      - !Ref SubnetA
      - !Ref SubnetB
    DependsOn: 'VPCGatewayAttachment'
```

The connection between the ALB and the auto-scaling group is made in the auto-scaling group description by specifying `TargetGroupARNs`.

The full CloudFormation template is located at http://mng.bz/S6Sj. Create a stack based on that template by clicking on the Quick-Create link at http://mng.bz/lbG4, and then visit the output of your stack with your browser. Every time you reload the page, you should see one of the private IP addresses of a backend web server.

To get some detail about the load balancer in the graphical user interface, navigate to the EC2 Management Console. The sub-navigation menu on the left has a Load Balancing section where you can find a link to your load balancers. Select the one and only load balancer. You will see details at the bottom of the page. The details contain a Monitoring tab, where you can find charts about latency, number of requests, and much more. Keep in mind that those charts are one minute behind, so you may have to wait until you see the requests you made to the load balancer.

Cleaning up

Delete the CloudFormation stack you created.

15.2 Asynchronous decoupling with message queues

Synchronous decoupling with ELB is easy; you don't need to change your code to do it. But for asynchronous decoupling, you have to adapt your code to work with a message queue.

A message queue has a head and a tail. You can add new messages to the tail while reading messages from the head. This allows you to decouple the production and consumption of messages. Decoupling the producers/requesters from consumers/receivers delivers with the following benefits:

- *The queue acts as a buffer*—Producers and consumers don't have to run at the same speed. For example, you can add a batch of 1,000 messages in one minute while your consumers always process 10 messages per second. Sooner or later, the consumers will catch up and the queue will be empty again.
- *The queue hides your backend*—Similar to the load balancer, messages producers have no knowledge of the consumers. You can even stop all consumers and still produce messages. This is handy while doing maintenance on your consumers.

The producers and consumers don't know each other; they both only know about the message queue. Figure 15.4 illustrates this principle.

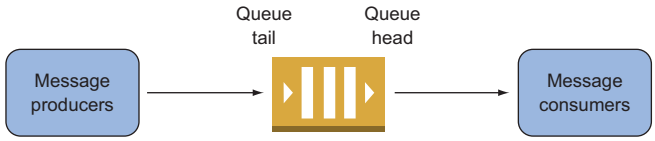


Figure 15.4 Producers send messages to a message queue while consumers read messages.

You can put new messages into the queue while no one is reading messages, and the message queue acts as a buffer. To prevent message queues from growing infinitely large, messages are only saved for a certain amount of time. If you consume a message from a message queue, you must acknowledge the successful processing of the message to permanently delete it from the queue.

The *Simple Queue Service* (SQS) is a fully managed AWS service. SQS offers message queues that guarantee the delivery of messages at least once:

- Under rare circumstances, a single message will be available for consumption twice. This may sound strange if you compare it to other message queues, but you'll see how to deal with this problem later in the chapter.
- SQS doesn't guarantee the order of messages, so you may read messages in a different order than they were produced.

This limitation of SQS is also beneficial:

- You can put as many messages into SQS as you like.
- The message queue scales with the number of messages you produce and consume.
- SQS is highly available by default.
- You pay per message.

The pricing model is also simple: you pay $0.00000040 USD per request to SQS or $0.4 USD per million requests. Producing a message is one request, and consuming is another request (if your payload is larger than 64 KB, every 64 KB chunk counts as one request).

15.2.1 Turning a synchronous process into an asynchronous one

A typical synchronous process looks like this: a user makes a request to your web server, something happens on the web server, and a result is returned to the user. To make things more concrete, we'll talk about the process of creating a preview image of an URL in the following example:

1. The user submits a URL.
2. The web server downloads the content at the URL, takes a screenshot, and renders it as a PNG image.
3. The web server returns the PNG to the user.

With one small trick, this process can be made asynchronous, and benefit from the elasticity of a message queue, for example during peak traffic:

1. The user submits a URL.
2. The web server puts a message into a queue that contains a random ID and the URL.
3. The web server returns a link to the user where the PNG image will be found in the future. The link contains the random ID (such as http://$Bucket.s3-website-us-east-1.amazonaws.com/$RandomId.png).
4. In the background, a worker consumes the message from the queue, downloads the content, converts the content into a PNG, and uploads the image to S3.
5. At some point, the user tries to download the PNG at the known location. If the file is not found, the user should reload the page in a few seconds.

If you want to make a process asynchronous, you must manage the way the process initiator tracks the process status. One way of doing that is to return an ID to the initiator that can be used to look up the process. During the process, this ID is passed from step to step.

15.2.2 Architecture of the URL2PNG application

You'll now create a simple but decoupled piece of software named URL2PNG that renders a PNG from a given web URL. You'll use Node.js to do the programming part, and you'll use SQS as the message queue implementation. Figure 15.5 shows how the URL2PNG application works.

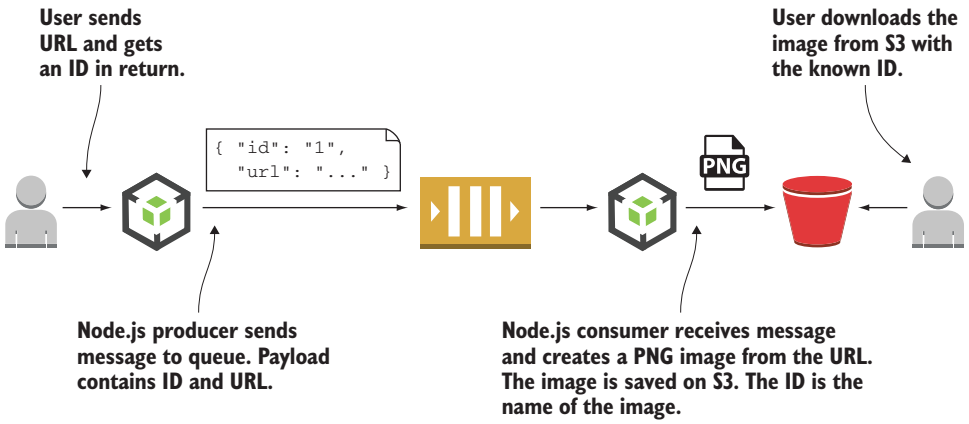


Figure 15.5 Node.js producer sends a message to the queue. The payload contains an ID and URL.

On the message producer side, a small Node.js scripts generates a unique ID, sends a message to the queue with the URL and ID as the payload, and returns the ID to the user. The user now starts checking if a file is available on the S3 bucket using the returned ID as the filename.

Simultaneously, on the message consumer side, a small Node.js script reads a message from the queue, generates the screenshot of the URL from the payload, and uploads the resulting image to an S3 bucket using the unique ID from the payload as the filename.

To complete the example, you need to create an S3 bucket with web hosting enabled. Execute the following commands, replacing *$yourname* with your name or nickname to prevent name clashes with other readers (remember that S3 bucket names have to be globally unique across all AWS accounts):

```
$ aws s3 mb s3://url2png-$yourname
$ aws s3 website s3://url2png-$yourname --index-document index.html \
➥ --error-document error.html
```

Web hosting is needed so users can later download the images from S3. Now it's time to create the message queue.

15.2.3 Setting up a message queue

Creating an SQS queue is simple: you only need to specify the name of the queue:

```
$ aws sqs create-queue --queue-name url2png
{
  "QueueUrl": "https://queue.amazonaws.com/878533158213/url2png"
}
```

The returned `QueueUrl` is needed later in the example, so take a note.

15.2.4 Producing messages programmatically

You now have an SQS queue to send messages to. To produce a message, you need to specify the queue and a payload. You'll use Node.js in combination with the AWS SDK to make requests to AWS.

> **Installing and getting started with Node.js**
>
> Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit https://nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 8.
>
> After Node.js is installed, you can verify if everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v8.*`. Now you're ready to run JavaScript examples like URL2PNG.
>
> Do you want to get started with Node.js? We recommend *Node.js in Action (2nd edition)* from Alex Young, et al., (Manning, 2017) or the video course *Node.js in Motion* from P.J. Evans, (Manning, 2018).

Here's how the message is produced with the help of the AWS SDK for Node.js; it will later be consumed by the URL2PNG worker. The Node.js script can then be used like this (don't try to run this command now—you need to install and configure URL2PNG first):

```
$ node index.js "http://aws.amazon.com"
PNG will be available soon at
http://url2png-$yourname.s3-website-us-east-1.amazonaws.com/XYZ.png
```

As usual, you'll find the code in the book's code repository on GitHub https://github.com/AWSinAction/code2. The URL2PNG example is located at /chapter15/url2png/. Here's the implementation of index.js.

Listing 15.2 index.js: sending a message to the queue

```
const AWS = require('aws-sdk');
const uuid = require('uuid/v4');
const sqs = new AWS.SQS({                          <-- Creates an SQS client
  region: 'us-east-1'
});

if (process.argv.length !== 3) {                   <-- Checks whether a URL was provided
  console.log('URL missing');
  process.exit(1);
}

const id = uuid();                                 <-- Creates a random ID
const body = {                                     <-- The payload contains the random ID and the URL.
  id: id,
  url: process.argv[2]
};

sqs.sendMessage({                                  <-- Converts the payload into a JSON string
  MessageBody: JSON.stringify(body),
  QueueUrl: '$QueueUrl'                            <-- Queue to which the message is sent (was returned when creating the queue).
}, (err) => {                                      <-- Invokes the sendMessage operation on SQS
  if (err) {
    console.log('error', err);
  } else {
    console.log('PNG will be soon available at ...');
  }
});
```

Before you can run the script, you need to install the Node.js modules. Run `npm install` in your terminal to install the dependencies. You'll find a config.json file that needs to be modified. Make sure to change *QueueUrl* to the queue you created at the beginning of this example, and change *Bucket* to `url2png-`*$yourname*.

Now you can run the script with `node index.js "http://aws.amazon.com"`. The program should respond with something like "PNG will be available soon at http://url2png-$yourname.s3-website-us-east-1.amazonaws.com/XYZ.png". To verify that the message is ready for consumption, you can ask the queue how many messages are inside. Replace `$QueueUrl` with your queue's URL.

```
$ aws sqs get-queue-attributes \
➥ --queue-url "$QueueUrl" \
➥ --attribute-names ApproximateNumberOfMessages
```

```
{
  "Attributes": {
    "ApproximateNumberOfMessages": "1"
  }
}
```

SQS only returns an approximation of the number of messages. This is due to the distributed nature of SQS. If you don't see your message in the approximation, run the command again and eventually you will see your message.

Next, it's time to create the worker that consumes the message and does all the work of generating a PNG.

15.2.5 Consuming messages programmatically

Processing a message with SQS takes three steps:

1. Receive a message.
2. Process the message.
3. Acknowledge that the message was successfully processed.

You'll now implement each of these steps to change a URL into a PNG.

To receive a message from an SQS queue, you must specify the following:

- `QueueUrl`—The unique queue identifier.
- `MaxNumberOfMessages`—The maximum number of messages you want to receive (from 1 to 10). To get higher throughput, you can get messages in a batch. We usually set this to 10 for best performance and lowest overhead.
- `VisibilityTimeout`—The number of seconds you want to remove this message from the queue to process it. Within that time, you must delete the message, or it will be delivered back to the queue. We usually set this to the average processing time multiplied by four.
- `WaitTimeSeconds`—The maximum number of seconds you want to wait to receive messages if they're not immediately available. Receiving messages from SQS is done by polling the queue. But AWS allows long polling, for a maximum of 10 seconds. When using long polling, you will not get an immediate response from the AWS API if no messages are available. If a new message arrives within 10 seconds, the HTTP response will be sent to you. After 10 seconds, you also get an empty response.

Ths listing shows how this is done with the SDK.

Listing 15.3 worker.js: receiving a message from the queue

```
const fs = require('fs');
const AWS = require('aws-sdk');
const webshot = require('webshot');
const sqs = new AWS.SQS({
  region: 'us-east-1'
```

```
});
const s3 = new AWS.S3({
  region: 'us-east-1'
});

const receive = (cb) => {
  const params = {
    QueueUrl: '$QueueUrl',
    MaxNumberOfMessages: 1,
    VisibilityTimeout: 120,
    WaitTimeSeconds: 10
  };
  sqs.receiveMessage(params, (err, data) => {
    if (err) {
      cb(err);
    } else {
      if (data.Messages === undefined) {
        cb(null, null);
      } else {
        cb(null, data.Messages[0]);
      }
    }
  });
};
```

Consumes no more than one message at once

Takes the message from the queue for 120 seconds

Long poll for 10 seconds to wait for new messages

Invokes the receiveMessage operation on SQS

Checks whether a message is available

Gets the one and only message

The receive step has now been implemented. The next step is to process the message. Thanks to the Node.js module `webshot`, it's easy to create a screenshot of a website.

Listing 15.4 worker.js: processing a message (take screenshot and upload to S3)

```
const process = (message, cb) => {
  const body = JSON.parse(message.Body);
  const file = body.id + '.png';
  webshot(body.url, file, (err) => {
    if (err) {
      cb(err);
    } else {
      fs.readFile(file, (err, buf) => {
        if (err) {
          cb(err);
        } else {
          const params = {
            Bucket: 'url2png-$yourname',
            Key: file,
            ACL: 'public-read',
            ContentType: 'image/png',
            Body: buf
          };
          s3.putObject(params, (err) => {
            if (err) {
              cb(err);
            } else {
              fs.unlink(file, cb);
            }
```

The message body is a JSON string. You convert it back into a JavaScript object.

Creates the screenshot with the webshot module

Opens the screenshot that was saved to local disk by the webshot module

Allows everyone to read the screenshot on S3

Uploads the screenshot to S3

Removes the screenshot from local disk

```
          });
        }
      });
    }
  });
};
```

The only step that's missing is to acknowledge that the message was successfully consumed. This is done by deleting the message from the queue after successfully completing the task. If you receive a message from SQS, you get a `ReceiptHandle`, which is a unique ID that you need to specify when you delete a message from a queue.

Listing 15.5 worker.js: acknowledging a message (deletes the message from the queue)

```
const acknowledge = (message, cb) => {
  const params = {
    QueueUrl: '$QueueUrl',
    ReceiptHandle: message.ReceiptHandle
  };
  sqs.deleteMessage(params, cb);
};
```

ReceiptHandle is unique for each receipt of a message.

Invokes the deleteMessage operation on SQS

You have all the parts; now it's time to connect them.

Listing 15.6 worker.js: connecting the parts

```
const run = () => {
  receive((err, message) => {
    if (err) {
      throw err;
    } else {
      if (message === null) {
        console.log('nothing to do');
        setTimeout(run, 1000);
      } else {
        console.log('process');
        process(message, (err) => {
          if (err) {
            throw err;
          } else {
            acknowledge(message, (err) => {
              if (err) {
                throw err;
              } else {
                console.log('done');
                setTimeout(run, 1000);
              }
            });
          }
        });
      }
    }
```

Receives a message

Checks whether a message is available

Calls the run method again in one second

Processes the message

Acknowledges the message

Calls the run method again in one second to poll for further messages (kind of a recursive loop, but with a timer in between. When the timer starts, a new call stack is allocated; this will not lead to a stack overflow!).

```
  });
};

run();                    <-- Calls the run method to start
```

Now you can start the worker to process the message that is already in the queue. Run the script with `node worker.js`. You should see some output that says the worker is in the process step and then switches to Done. After a few seconds, the screenshot should be uploaded to S3. Your first asynchronous application is complete.

Remember the output you got when you invoked `node index.js "http://aws.amazon.com"` to send a message to the queue? It looked similar to this: http://url2png-$yourname.s3-website-us-east-1.amazonaws.com/XYZ.png. Now put that URL in your web browser and you will find a screenshot of the AWS website (or whatever you used as an example).

You've created an application that is asynchronously decoupled. If the URL2PNG service becomes popular and millions of users start using it, the queue will become longer and longer because your worker can't produce that many PNGs from URLs. The cool thing is that you can add as many workers as you like to consume those messages. Instead of only one worker, you can start 10 or 100. The other advantage is that if a worker dies for some reason, the message that was in flight will become available for consumption after two minutes and will be picked up by another worker. That's fault-tolerant! If you design your system to be asynchronously decoupled, it's easy to scale and a good foundation to be fault-tolerant. The next chapter will concentrate on this topic.

Cleaning up

Delete the message queue as follows:

```
$ aws sqs delete-queue --queue-url "$QueueUrl"
```

And don't forget to clean up and delete the S3 bucket used in the example. Issue the following command, replacing *$yourname* with your name:

```
$ aws s3 rb --force s3://url2png-$yourname
```

15.2.6 Limitations of messaging with SQS

Earlier in the chapter, we mentioned a few limitations of SQS. This section covers them in more detail. But before we start with the limitations, here are the benefits:

- You can put as many messages into SQS as you like. SQS scales the underlying infrastructure for you.
- SQS is highly available by default.
- You pay per message.

Those benefits come with some trade-offs. Let's have a look of the limitations in more detail now.

SQS DOESN'T GUARANTEE THAT A MESSAGE IS DELIVERED ONLY ONCE

There are two reasons why a message might be delivered more than once:

1. Common reason: If a received message isn't deleted within `VisibilityTimeout`, the message will be received again.
2. Rare reason: If a `DeleteMessage` operation doesn't delete all copies of a message because one of the servers in the SQS system isn't available at the time of deletion.

The problem of repeated delivery of a message can be solved by making the message processing idempotent. *Idempotent* means that no matter how often the message is processed, the result stays the same. In the URL2PNG example, this is true by design: If you process the message multiple times, the same image will be uploaded to S3 multiple times. If the image is already available on S3, it's replaced. Idempotence solves many problems in distributed systems that guarantee messages will be delivered at least once.

Not everything can be made idempotent. Sending an email is a good example: if you process a message multiple times and it sends an email each time, you'll annoy the addressee.

In many cases, processing at least once is a good trade-off. Check your requirements before using SQS if this trade-off fits your needs.

SQS DOESN'T GUARANTEE THE MESSAGE ORDER

Messages may be consumed in a different order than the order in which you produced them. If you need a strict order, you should search for something else. SQS is a fault-tolerant and scalable message queue. If you need a stable message order, you'll have difficulty finding a solution that scales like SQS. Our advice is to change the design of your system so you no longer need the stable order, or put the messages in order on the client side.

> **SQS FIFO (first-in-first-out) queues**
>
> FIFO queues guarantee order of messages and have a mechanism to detect duplicate messages. If you need a strict message order, they are worth a look. The disadvantages are higher pricing and a limitation on 300 operations per second. Check out the documentation at http://mng.bz/Y5KN for more information.

SQS DOESN'T REPLACE A MESSAGE BROKER

SQS isn't a message broker like ActiveMQ—SQS is only a message queue. Don't expect features like message routing or message priorities. Comparing SQS to ActiveMQ is like comparing DynamoDB to MySQL.

Amazon MQ

AWS announced an alternative to Amazon SQS in November 2017: Amazon MQ provides Apache ActiveMQ as a service. Therefore, you can use Amazon MQ as a message broker that speaks the JMS, NMS, AMQP, STOMP, MQTT, and WebSocket protocols.

Go to the Amazon MQ Developer Guide at https://docs.aws.amazon.com/amazon-mq/latest/developer-guide/ to learn more.

Summary

- Decoupling makes things easier because it reduces dependencies.
- Synchronous decoupling requires two sides to be available at the same time, but the sides don't have to know each other.
- With asynchronous decoupling, you can communicate without both sides being available.
- Most applications can be synchronously decoupled without touching the code, by using a load balancer offered by the ELB service.
- A load balancer can make periodic health checks to your application to determine whether the backend is ready to serve traffic.
- Asynchronous decoupling is only possible with asynchronous processes. But you can modify a synchronous process to be an asynchronous one most of the time.
- Asynchronous decoupling with SQS requires programming against SQS with one of the SDKs.